Compilers

Dr. Sherin ElGokhy

Lecture#4

Introduction to Parsing

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFG's)
- Derivations
- Ambiguity

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages, tree languages

Beyond Regular Languages

- Many languages are not regular
- Strings of balanced parentheses are not regular:

$$\left\{ (^{i})^{i} \mid i \geq 0 \right\}$$

What Can Regular Languages Express?

- Languages requiring counting modulo a fixed integer
- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can't remember # of times it has visited a particular state

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program

Example

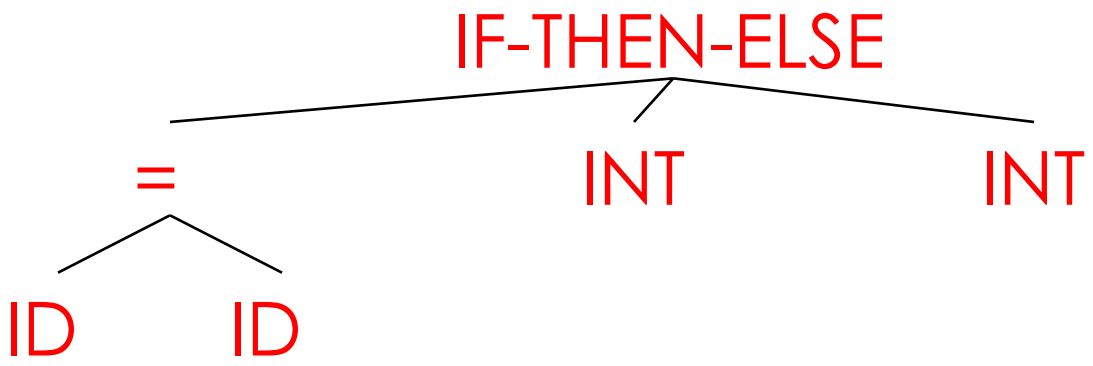
- Cool

  if x = y then 1 else 2 fi

- Parser input

  IF ID = ID THEN INT ELSE INT FI

- Parser output



Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

The Role of the Parser

- Not all strings of tokens are valid programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens

- We need
  - A language for **describing valid strings of tokens**
  - A method for **distinguishing valid from invalid strings of tokens**

Context-Free Grammars

- Programming language constructs have recursive structure
- Expressions are themselves recursively composed of other expressions.

- An EXPR is
  - if EXPR then EXPR else EXPR fi
  - while EXPR loop EXPR pool
  - …

- Context-free grammars are a natural notation for describing this recursive structure

CFGs (Cont.)

- A CFG consists of
  - A set of *terminals* T, *alphabet of the language*
  - A set of *non-terminals* N
  - A *start symbol* S (a non-terminal)
  - A set of *productions*

$$X \rightarrow Y_1 Y_2 \cdots Y_n$$

where $X \in N$ and $Y_i \in T \cup N \cup \{\varepsilon\}$

Notational Conventions

Notes

- Non-terminals are written upper-case
- Terminals are written lower-case
- The start symbol is the left-hand side of the first production

Examples of CFGs

EXPR	→	if EXPR then EXPR else EXPR fi
	|	while EXPR loop EXPR pool
	|	id

Examples of CFGs (cont.)

Simple arithmetic expressions:

$$
\begin{aligned}
E &\rightarrow E * E \\
&\mid E + E \\
&\mid (E) \\
&\mid id
\end{aligned}
$$

The Language of a CFG

Read productions as rules:

$$X \rightarrow Y_1 \cdots Y_n$$

Means X can be replaced by $Y_1 \cdots Y_n$

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by the right-hand side of some production

$$X \rightarrow Y_1 \cdots Y_n$$

3. Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

More formally, write

$$X_1 \cdots X_i \cdots X_n \rightarrow X_1 \cdots X_{i-1} Y_1 \cdots Y_m X_{i+1} \cdots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \cdots Y_m$$

The Language of a CFG (Cont.)

Write

$$X_1 \cdots X_n \overset{*}{\rightarrow} Y_1 \cdots Y_m$$

If in a number of steps>=0

$$X_1 \cdots X_n \rightarrow \cdots \rightarrow \cdots \rightarrow Y_1 \cdots Y_m$$

The Language of a CFG

Let *G* be a context-free grammar with start symbol S.
Then the language of *G* is:

$$\left\{ a_1 \ldots a_n \mid S \overset{*}{\rightarrow} a_1 \ldots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

S goes in zero or more steps to a string of terminals

Terminals

- Terminals are so-called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals must be tokens of the language

Examples

L(G) is the language of CFG G

Strings of balanced parentheses

$$\left\{ (^{i})^{i} \mid i \geq 0 \right\}$$

Two grammars:

$$S \rightarrow (S)$$
$$S \rightarrow \varepsilon$$

OR

$$S \rightarrow (S) \mid \varepsilon$$

Cool Example

A fragment of COOL:

EXPR	→	if EXPR then EXPR else EXPR fi
	|	while EXPR loop EXPR pool
	|	id

Cool Example (Cont.)

Some elements of the language

id

if id then id else id fi

while id loop id pool

if while id loop id pool then id else id

if if id then id else id fi then id else id fi

Arithmetic Example

Simple arithmetic expressions:

$$E \rightarrow E+E \mid E * E \mid (E) \mid id$$

Some elements of the language:

id	id + id
(id)	id * id
(id) * id	id * (id)

Quiz

Which of the strings are in the language of the given CFG?

- ☐ abcba
- ☐ acca
- ☐ aba
- ☐ abcbcba

$$
\begin{aligned}
S &\rightarrow aXa \\
X &\rightarrow \varepsilon \\
&\mid bY \\
Y &\rightarrow \varepsilon \\
&\mid cXc
\end{aligned}
$$

Notes

The idea of a CFG is a big step. But, we still need some other things.

- Membership in a language is "yes" or "no"; also need parse tree of the input
- Must handle errors
- Need an implementation of CFG's (e.g., bison)

More Notes

- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar
  - Note: Tools for regular languages (e.g., flex) are sensitive to the form of the regular expression, but this is rarely a problem in practice

Derivations and Parse Trees

A *derivation* is a sequence of productions

$$S \rightarrow \cdots \rightarrow \cdots \rightarrow \cdots$$

A derivation can be drawn as a tree

- Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \cdots Y_n$ add children $Y_1 \cdots Y_n$ to node X

Derivation Example

- Grammar

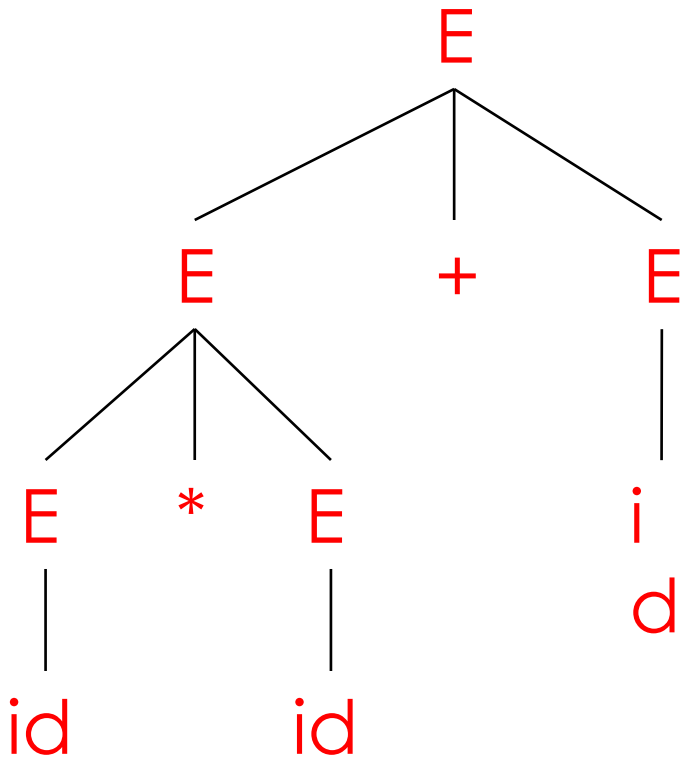
$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

- String

$$id * id + id$$

Derivation Example (Cont.)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Derivation in Detail (1)

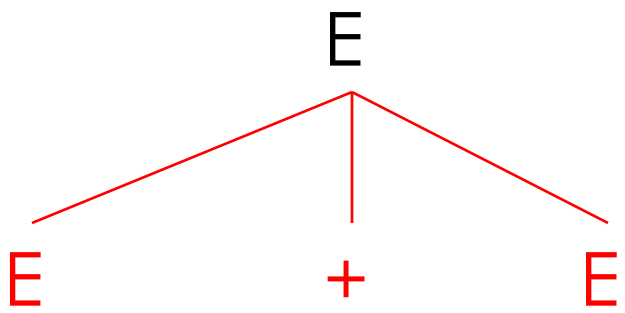
E

E

Derivation in Detail (2)

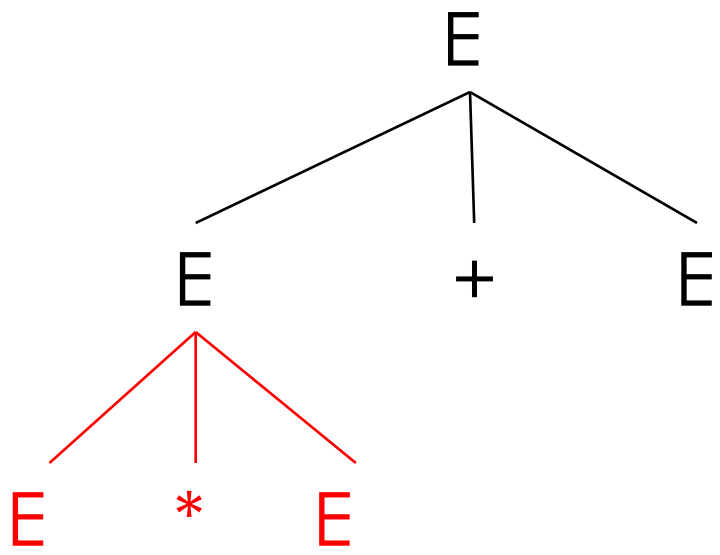
E

→ E+E



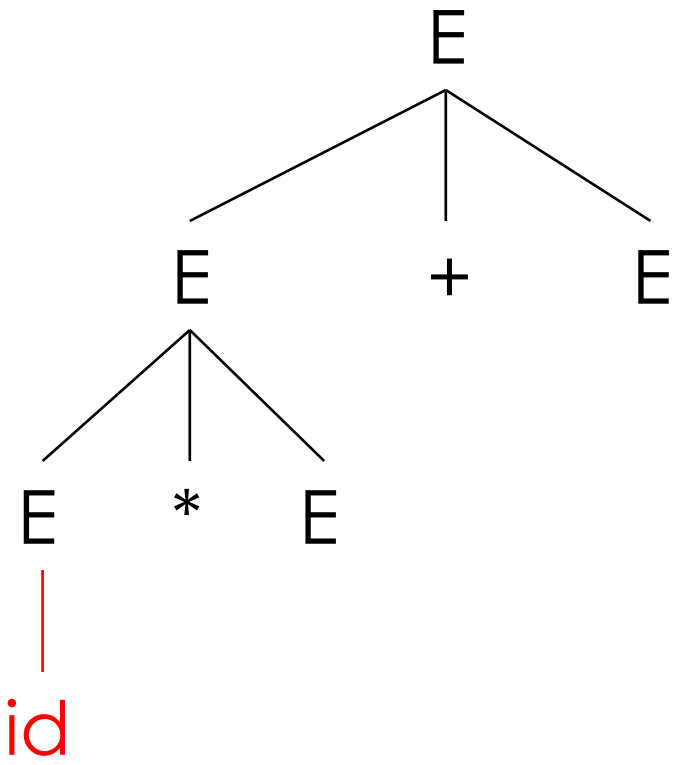
Derivation in Detail (3)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E*E+E$$



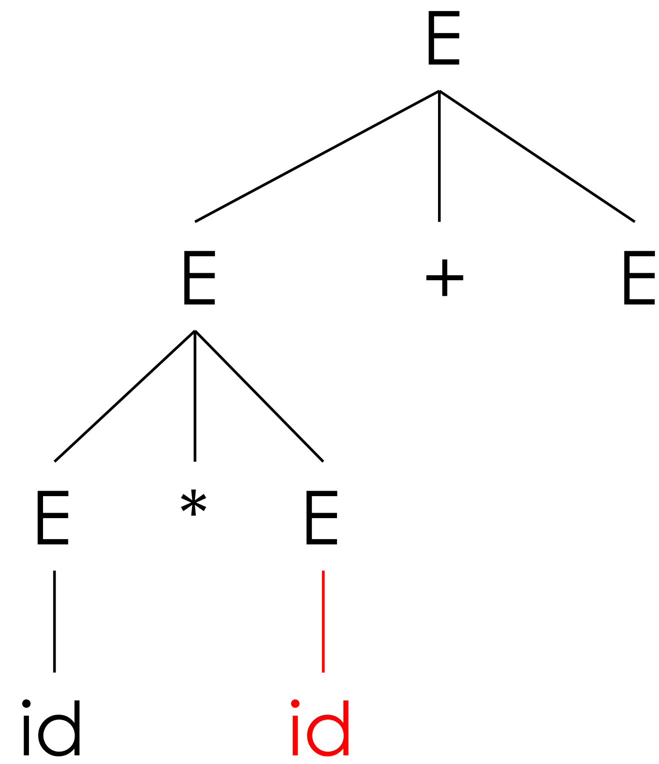
Derivation in Detail (4)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E
\end{aligned}
$$



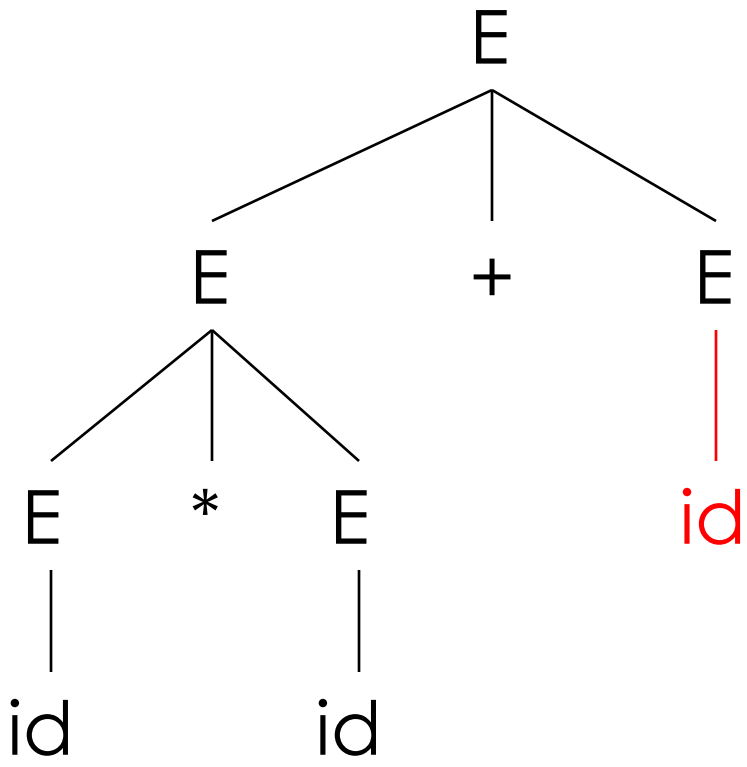
Derivation in Detail (5)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E*E+E$$
$$\rightarrow id*E+E$$
$$\rightarrow id*id+E$$



Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations, the input string does not

Left-most and Right-most Derivations

- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E*E+id \\
\rightarrow\ & E*id+id \\
\rightarrow\ & id*id+id
\end{aligned}
$$

Right-most Derivation in Detail (1)

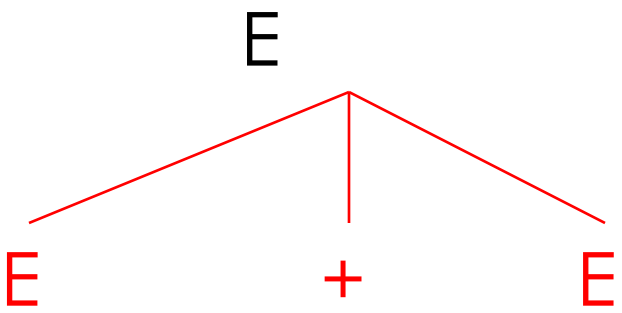
E

E

Right-most Derivation in Detail (2)

E

→ E+E

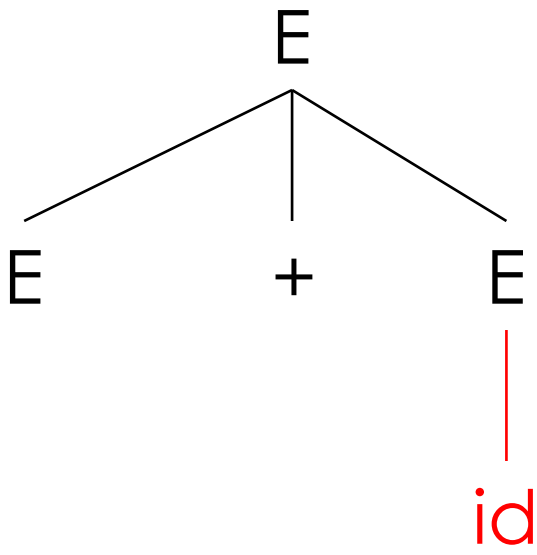


Right-most Derivation in Detail (3)

E

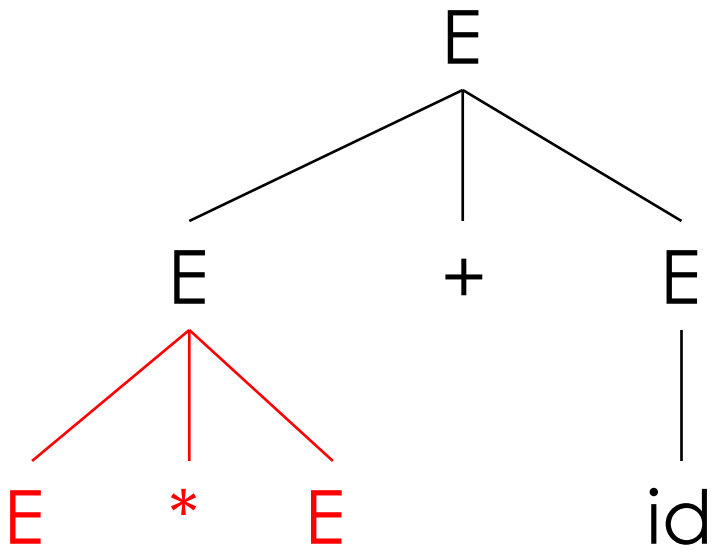
$\rightarrow$ E+E

$\rightarrow$ E+id



Right-most Derivation in Detail (4)

$$E$$
$$\rightarrow E+E$$
$$\rightarrow E+id$$
$$\rightarrow E*E+id$$



Right-most Derivation in Detail (5)

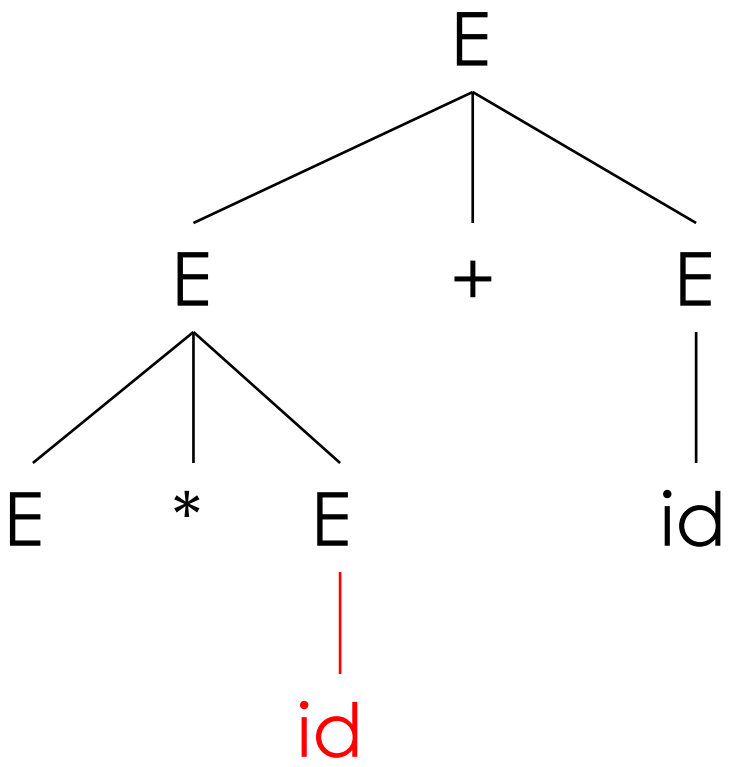
E

$\rightarrow E+E$

$\rightarrow E+id$

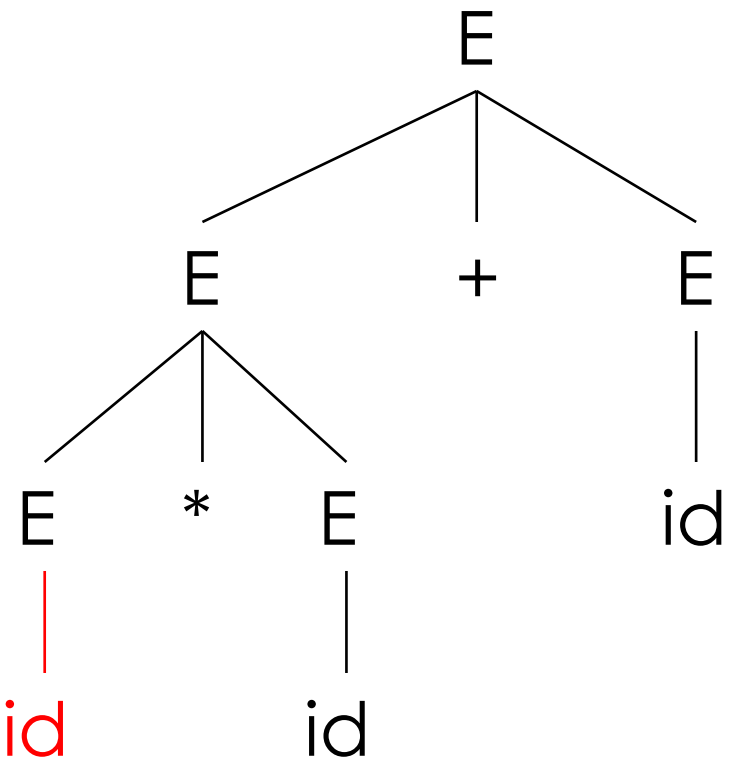
$\rightarrow E*E+id$

$\rightarrow E*id+id$



Right-most Derivation in Detail (6)

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E*E+id \\
\rightarrow\ & E*id+id \\
\rightarrow\ & id*id+id
\end{aligned}
$$



Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Quiz

Which of the following is a valid derivation of the given grammar?

$S \rightarrow aXa$

$X \rightarrow \varepsilon \mid bY$

$Y \rightarrow \varepsilon \mid cXc \mid d$

- S
  aXa
  abYa
  acXca
  acca
- S
  aXa
  abYa
  abcXca
  abcbYca
  abcbdca
- S
  aa
- S
  aXa
  abYa
  abcXcda
  abccda

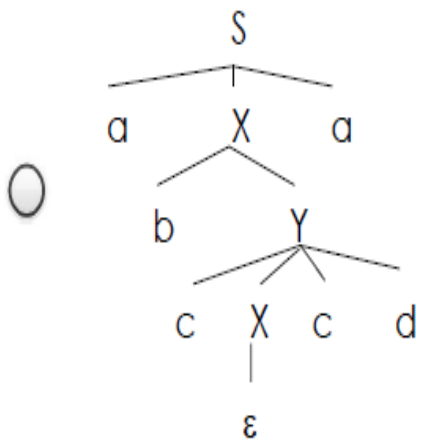
Quiz

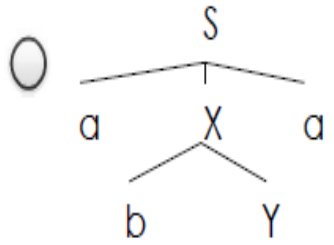
Which of the following is a valid parse tree for the given grammar?

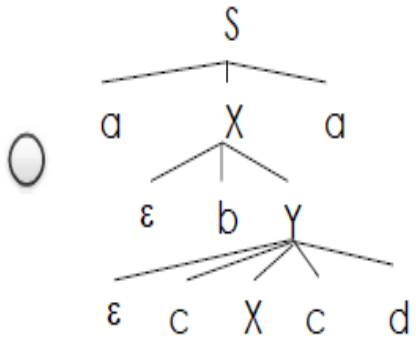
$S \rightarrow aXa$

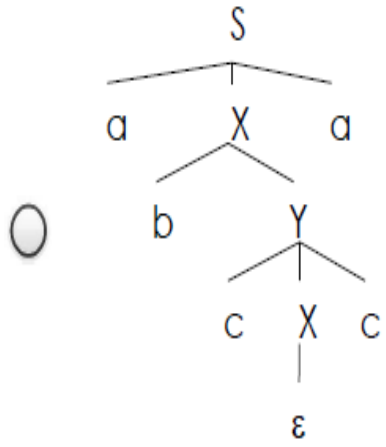
$X \rightarrow \varepsilon \mid bY$

$Y \rightarrow \varepsilon \mid cXc \mid d$









Summary of Derivations

- We are not just interested in whether $s \varepsilon L(G)$
  - We need a parse tree for *s*
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar

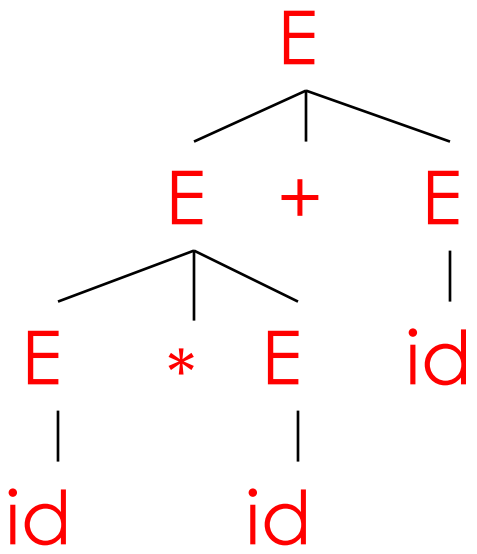
$$E \rightarrow E+E \mid E*E \mid (E) \mid id$$

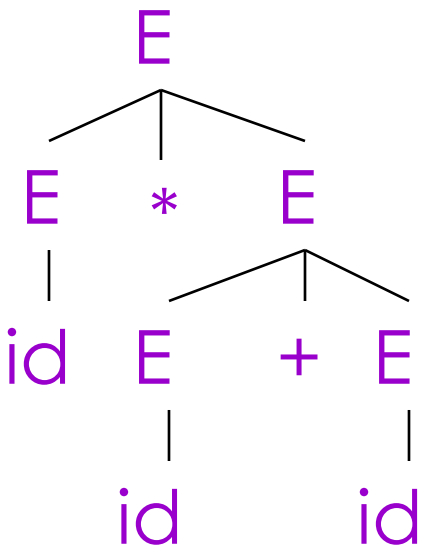
- String

$$id * id + id$$

Ambiguity (Cont.)

This string has two different parse trees for the same string





Ambiguity (Cont.)

- A grammar is *ambiguous* if it has more than one parse tree for some string……distinct parse trees
  - Equivalently, there is more than one right-most or left-most derivation for some string
- Ambiguity is **BAD**
  - Leaves meaning of some programs ill-defined

Quiz

Which of the following grammars are ambiguous?

- ☐ $S \rightarrow SS \mid a \mid b$
- ☐ $E \rightarrow E + E \mid id$
- ☐ $S \rightarrow Sa \mid Sb$
- ☐ $E \rightarrow E' \mid E' + E$

  $E' \rightarrow -E' \mid id \mid (E)$

Thanks